Project 1[1]

Due by 11:59pm 9/26/2018

Objective

This assignment explores the connection between problem formulation and search strategies. You will need to implement and compare several search strategies on a few different puzzles and problems we've talked about in class.

Program Specification

Your program should be named "puzzlesolver.py" and it should take *two mandatory input arguments plus one optional one*. The first argument is a configuration file for the puzzle your program has to solve (see the next section for more information). The second argument is a keyword that specifies which search algorithm to use to solve the puzzle: **bfs**, **dfs**, **unicost,** **greedy,** and **astar**. The optional third argument allows the user to specify different heuristic functions for **greedy** and **astar** search.

Your program should solve the input puzzle using the specified search algorithm and strategy. Upon completion, it should output the following information:

- the solution path (or "No solution" if nothing was found): print one state per line.
- **Time**: expressed in terms of the total number of nodes created.
- **Space**: expressed as the *maximum* number of nodes kept in memory (approximately,, the biggest size that the frontier list grew to) as well as the maximum number of states stored on the explored list (if using). Please report these as two separate numbers.

Puzzle Types

An important element of this assignment is to separate the puzzle specific parts from the generic search algorithm part. Make sure that your implementation of the search does not hard code in any puzzle specific details. For this assignment, your program will be working with three puzzles: the water jug problem, the path planning puzzle, and the tile puzzle. If you've kept the problem/search abstraction clean, you ought to be able to reuse much of your infrastructure code. For each new puzzle type, should only need to modify the functions relevant to setting up its state space search (e.g., get-successor-states, and goal-test, etc.)

The Water Jugs Problem

We've previously talked about this problem in Lecture 2:

---

[1] This document is also available on Google Drive: https://docs.google.com/document/d/1nz_NSEqmjxZ1DRmHovCBI38h404rD05Un1NizYGMqyo/edit?usp=sharing

> "You are given two jugs, a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a tap that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?". (cf. Rich & Knight, 1991)

In general, we could have more than two jugs of some other pre-specified volumes (For example, see this Wikipedia entry for a version with 3 jugs). We will use the configuration file to specify the particular instance of the puzzle. It has the following format:

Line 1 is a keyword (**jugs**) that tells you this is the water jug problem
Line 2 is a tuple specifying the capacities of the jugs; e.g., (3, 4)
Line 3 is the initial state; e.g., (0, 0)
Line 4 is the goal state; e.g., (0, 2)

You may assume that we **will not** stress test with bad inputs like negative capacities or impossible states, and we'll keep to just two or three jugs.

The Path-Planning Problem

For this problem, we will read in a graph representation of a roadmap, where cities are nodes and roads between them are undirected edges with costs. We want to find a path from the starting city to the destination city with the lowest cost. Here is the format of the input configuration file:

Line 1 is a keyword (**cities**) that tells you that this is the path-planning problem.
Line 2 is a list of the names and locations of the cities on some grid; e.g., [("Arlington",1,1),("Berkshire",2,3),("Chelmsford",1, 5)]
Line 3 is the name of the starting city
Line 4 is the name of the destination city
Each of the following lines specifies a connection between two cities and the cost of taking that route; e.g.,: ("Arlington", "Chelmsford", 4) specifies that it costs 4 units to go from Arlington to Chelmsford or vice versa (i.e.,after one action).

You may assume that we **will not** stress test with bad inputs like roads with negative or zero cost. We may try a large graph with this problem, however.

The Tile Puzzle

We have an NxN grid (recommended value: N=3). In each cell has a tile with some number between 1 and N^2-1 or a blank. We can slide a tile into the blank by moving it up, down, left, or right. The goal state is for all the numbers to go in sequence from the upper left to the lower right, with the blank being at the lower right corner. Here is the format of the input configuration:

Line 1 is a keyword (**tiles**) that tells you that this is the tiles problem.
Line 2 is an integer specifying N (in this example, 3)
Line 3 is the start game configuration: assume this describes the order of the tiles in rows from the upper left to the bottom right. 'b' means blank. Below is one
[8,6,7,2,5,4,3,'b',1]
Line 4 is the name of the goal state:
[1,2,3,4,5,6,7,8,'b']

NB: to generate the initial state, don't randomly permute the numbers because the goal state might not be reachable (the space of all possible random tile permutation forms two disconnected graphs). You can do the parity test as described in the Wikipedia page, if you like, but it's also fine to start from the goal state and make random moves to construct a reachable initial state (this would be a separate little support function for your own use. We won't grade this), or, just use our given sample as a test, though it's supposed to be a difficult one.

Sample Configuration Files

A sample input configuration file is provided for each puzzle type in the repository, but your program ought to be able to handle variations on the same puzzle family (e.g., if we change the capacities of the jugs, it shouldn't break your program).

Recommended Approach

- Begin by implementing the uninformed search strategies (**bfs**, **dfs**, and **unicost)** for the water jug and path planning problems.
- Next, implement the informed search algorithms (**greedy,** and **astar**). Try some simple heuristic functions for each puzzle type.
- Try to have fully debugged your program before extending it to cover the tile puzzle because it's probably easier to trace out what went wrong in the water jug cases than the tile puzzle case.
- For the tile puzzle, you should implement at least the Manhattan Distance heuristic (refer to Lecture 5 or 6). You may think of other ones if you like.
- Please make a final commit by 11:59pm 9/26 (Wednesday).

What to commit

- Your Python source code.
- A README file that contains the following information:
    - the version of Python you used
    - How to run your program -- for example, what keywords should we use to try out the different heuristic functions that you've implemented?
    - List any additional resources, references, or web pages you've consulted.
    - List any person with whom you've discussed the assignment and describe the nature of your discussions.
    - (final commit) Describe any component of your program that is not working
- (final commit) A transcript of your program running on the test cases that we will release next Wednesday. NB: on Linux or Mac Terminal or Cygwin on Windows, use the command "script" which will save your session into a text file. For DOS Prompt, you may have to find some workarounds (See: [1], [2]).
- (final commit) A report that discusses the relative performances of the search algorithms and the heuristic functions.
    - What heuristic function did you use for each puzzle class?

    - Did all the outcomes make sense (e.g., do the time/space complexities of different search strategies match your expectation based on our class discussions? What about optimality and completeness?)
    - Was there anything that surprised you? If so, elaborate.

Grading Guideline

Assignments are graded qualitatively on a non-linear five point scale. Below is a rough guideline:

- 5 (100%): The program works as expected; has a clear README for the grader; includes a transcript; complete report with insightful observations.
- 4 (93%): The program works as expected, but possibly with some minor flaws; has a clear enough README for the grader; includes a transcript; complete report.
- 3 (80%): The program works for all search strategies for at least two puzzle types, and there is a clear abstraction between the search and puzzle parts; OR, the program works for at least three search strategies for all three puzzle types; has a clear enough README for the grader; includes a transcript; cursory report.
- 2 (60%): The program works for at least three search strategies for at least two puzzle types; has a clear enough README for the grader; includes a transcript; cursory report. OR: has a better program but the report is missing.
- 1 (40%): The program has major problems, but a serious attempt has been made on the assignment; has a clear enough README for the grader.